
AIMS Documentation

Release 1.1.0

Daniel R. Reese

April 25, 2016

1	Contents	1
1.1	Project Summary	1
1.2	Acknowledgements	2
1.3	Requirements	2
1.4	Download AIMS	3
1.5	Usage	3
1.6	File formats	5
1.7	The AIMS program	8
1.8	The model module	19
1.9	The constants module	31
1.10	The utilities module	32
1.11	The ImprovedTessellation module	32
1.12	The plot_interpolation_test tool	34
2	Indices and tables	37
	Python Module Index	39
	Index	41

CONTENTS

1.1 Project Summary

1.1.1 Description

Name: “Astero seismic Inference on a Massive Scale” (*AIMS*)

Goals:

- estimate stellar parameters and credible intervals/error bars
- chose a representative set or sample of reference models
- be computationally efficient

Inputs:

- classic constraints and error bars (Teff, L, ...)
- seismic constraints and error bars (individual frequencies)

Requirements:

- a *precalculated* grid of models including:
 - the models themselves
 - parameters for the model (M, R, Teff, age, ...)
 - theoretical frequency spectra for the models

Methodology:

- applies an MCMC algorithm based on the python package [emcee](#). Relevant articles include:
 - [Bazot et al. \(2012, MNRAS 427, 1847\)](#)
 - [Gruberbauer et al. \(2012, ApJ 749, 109\)](#)
- interpolates within the grid of models using Delaunay tessellation (from the [scipy.spatial](#) package which is based on the [Qhull](#) library)
- modular approach: facilitates including contributions from different people

1.1.2 Contributors

Author:

- Daniel R. Reese

Comments, corrections & suggestions:

- [Diego Bossini](#)
- [Tiago L. Campante](#)
- [William J. Chaplin](#)
- [Hugo R. Coelho](#)
- [Guy R. Davies](#)
- [James S. Kuszlewicz](#)
- [Martin W. Long](#)
- [Mikkel N. Lund](#)
- [Andrea Miglio](#)

1.1.3 Supplementary material

- a more technical [overview](#) of AIMS
- a PDF version of this documentation may be downloaded [here](#)

1.1.4 Copyright information

- the AIMS project is distributed under the terms of the [GNU General Public License, version 3](#)
- a copy of of this license may be downloaded [here](#) and should also be included in `AIMS.tgz`

1.2 Acknowledgements

The “Asteroseismic Inference on a Massive Scale” (AIMS) project was developed at the University of Birmingham by Daniel R. Reese as one of the deliverables for the SPACEINN network. The [SPACEINN network](#) is funded by the European Community’s Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 312844.

1.2.1 Publications

If AIMS is used in any publication, the SPACEINN network kindly asks you to acknowledge the use of this software using a phrase such as the following:

“This article made use of AIMS, a software for fitting stellar pulsation data, developed in the context of the SPACEINN network, funded by the European Commission’s Seventh Framework Programme.”

1.3 Requirements

The following python packages are needed for AIMS:

- [dill](#)
- [emcee](#)
- [corner](#)

- note: this used to be called triangle in previous releases
- `numpy`
- `matplotlib`
- `multiprocessing`

1.4 Download AIMS

- Click [here](#) to download AIMS.
- The contents of this file may then be extracted via the command:

```
tar -zxvf AIMS.tgz
```

- This will lead to the creation of a folder called `AIMS` and a subfolder called `AIMS/doc`.
 - the `AIMS` folder contains the AIMS program; it is from this folder that AIMS is launched.
 - the `AIMS/doc` folder is where the documentation is generated. Typing `make html` within this folder will generate this web page in `AIMS/doc/_build/html/`. Typing `make latexpdf` will generate a pdf version of this documentation in `AIMS/doc/_build/latex/AIMS.pdf`.

1.5 Usage

There are three different ways of using AIMS:

1. generating a binary file with the grid of models (including names, global parameters, and pulsation frequencies).

Note: This step must be carried out before the following two steps as these require the above binary file to function correctly.

2. carrying out tests to evaluate the accuracy of the interpolation for a given grid of models.
3. finding the properties of an observed star thanks to its classic and seismic parameters.

The way AIMS is used is decided by the values given in the `AIMS_configure.py` file, which also contains a number of other control parameters. Extensive comments are included in this file to help the user know how to set the various parameters.

1.5.1 Generating a binary grid

Requirements:

- a grid of models, including the pulsation frequencies; the format for the files with the pulsation frequencies is described in `model.Model.read_file()`.
- a list with the paths and a set of global parameters for each model in the grid; the format this file is described in `model.Model_grid.read_model_list()`.

Relevant parameters in `AIMS_configure.py`:

- `write_data`: set this to `True` so that AIMS will write binary grid.
- `list_grid`: set this to the filename of the file with the list of paths and global parameters.
- `binary_grid`: set this to the filename of the file which will contain the binary data.

- `grid_params`: specify the parameters relevant to the grid (excluding age, which is dealt with separately). Different options can be found in the source to `model.Model.string_to_param()`.
- `npositive`: set this to `True` to only save modes with $n \geq 0$ in the binary file.

To run AIMS in this configuration, just type the following in a terminal window:

```
./AIMS.py
```

1.5.2 Testing the accuracy of the interpolation

Requirements:

- a binary grid of models as produced by AIMS

Relevant parameters in `AIMS_configure.py`:

- `write_data`: set this to `False` otherwise a binary grid will be produced, the interpolations tests will not be carried out.
- `test_interpolation`: set this to `True` so that AIMS will carry out the interpolation tests.
- `interpolation_file`: specify the name of the file in which to write the results from the interpolation test in binary format. These results can be plotted using `plot_interpolation_test.py`.

To run AIMS in this configuration, just type the following in a terminal window:

```
./AIMS.py
```

1.5.3 Characterising an observed star

Requirements:

- a binary grid of models as produced by AIMS
- a file with the observational data; the format for this file is similar to the format used for the [Asteroseismic Modeling Portal \(AMP\)](#) with some simplifications and is described below. It will be read by `AIMS.Likelihood.read_constraints()`

Relevant parameters in `AIMS_configure.py`:

- `write_data`: set this to `False`
- `test_interpolation`: set this to `False`
- most of the parameters in this file - see comments for details

To run AIMS in this configuration, just type the following in a terminal window:

```
./AIMS.py file_with_constraints
```

where `file_with_constraints` is the file with the observational constraints.

1.6 File formats

1.6.1 Format of a file with a list of models and properties:

Description:

- The first line is a header. It contains the root folder (including the final slash) with the grid of models and optionally, a suffix for the names of the files with the theoretical pulsation frequencies. For example:

```
/home/dreese/models_inversions/Grid_mesa_MS/ .freq
```

- Each of the following lines correspond to one model in the grid. They are composed of 8 or more columns with the following information:
 1. The second part of the path for the given model. When concatenated with the prefix on the first line, this should give the full path to the model. If, furthermore, the suffix from the first line is appended to it, it gives the name of the file with the frequencies.
 2. The stellar mass in g
 3. The stellar radius in cm
 4. The stellar luminosity in $\text{g.cm}^2.\text{s}^{-3}$
 5. The metallicity
 6. The hydrogen content
 7. The stellar age in Myrs
 8. The effective temperature in K
 9. (user-defined) This and the following columns correspond to the parameters specified in the `user_params` variable given in `AIMS_configure.py`.
- Except for the first line, the order of the lines does not matter. AIMS will construct evolutionary tracks based on the parameters selected in the `grid_params` variable given in `AIMS_configure.py`, and sort them according to age.

Example: Here's an example of a file read by AIMS (via the `model.Model_grid.read_model_list()` method):

```
/home/dreese/models_inversions/Grid_mesa_MS/ .freq
M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n1.profile.FGONG
M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n10.profile.FGONG
M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536Z0.0028a1.8ovh0.2ovhe0_n11.profile.FGONG
```

It contains three models. The structure of the first model can be found in the following file:

```
/home/dreese/models_inversions/Grid_mesa_MS/M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536
```

and its frequencies in this file:

```
/home/dreese/models_inversions/Grid_mesa_MS/M0.80/LOGS_M0.80/M0.80Z0.0028Y0.2536/m0.80Y0.2536
```

The ninth column corresponds to the central hydrogen content, as specified by the contents of the `user_params` variable from `AIMS_configure.py`:

```
user_params = (("Xc", r'Central hydrogen, %sX_c%s'),)
```

1.6.2 Format of a file with theoretical frequencies:

Description:

- the first line is a header (and is skipped)
- the following lines contain five columns which correspond to l, n, frequency, a_value, inertia
 - the a_value column is ignored, so it could contain anything. `InversionKit` will typically put the difference between the numerical and variational frequencies in that column.

Example: Here’s an example of a file with theoretical pulsation frequencies which can be read by AIMS (via the `model.Model.read_file()` method):

#l	n	nu_theo (muHz)	nu_var-nu_theo (muHz)	Inertia
0	15	3.225852209451052e+03	1.312960435370769e-03	3.233628965187502e-09
0	16	3.421699035498995e+03	-2.482639610207116e-03	2.229252226305757e-09
0	17	3.615805033992529e+03	3.993051574070705e-03	1.618154348529283e-09
0	18	3.809740380503104e+03	9.650666734160040e-04	1.250359548964621e-09
0	19	4.003716857281849e+03	-7.991676880010345e-03	1.033914933206195e-09
0	20	4.198691419457581e+03	1.742711681799847e-03	8.866985261874711e-10
1	15	3.316007619955153e+03	5.056100344972947e-03	2.715966891128009e-09
1	16	3.511258977705781e+03	1.855844971032639e-04	1.902147334986236e-09
1	17	3.705576731149742e+03	-2.505276897409203e-03	1.424266453221534e-09
1	18	3.899485457373566e+03	5.21227655539575e-03	1.134594720287415e-09
1	19	4.094401244305849e+03	6.020260397235688e-03	9.579611596023003e-10
1	20	4.289716814475406e+03	-1.019475706561934e-02	8.344804874142957e-10
2	15	3.399280335063532e+03	-8.466318249702454e-04	2.315947651745295e-09
2	16	3.594141943503532e+03	4.712417365681176e-03	1.665322627996223e-09
2	17	3.788792185755381e+03	-1.167229517704982e-03	1.277569745555387e-09
2	18	3.983271067684743e+03	-6.187409578615188e-03	1.048757367028520e-09
2	19	4.178866833517976e+03	6.893199766636826e-03	8.963691946280509e-10
2	20	4.374959711016754e+03	3.274638356742798e-03	7.911508926344487e-10
3	15	3.476224140192640e+03	-2.524210208321165e-03	2.009476926536794e-09
3	16	3.671438520072859e+03	2.351724720028869e-04	1.485336526791650e-09
3	17	3.866350877376991e+03	5.643782460992952e-03	1.167619144668003e-09
3	18	4.061929209725198e+03	-1.552865011490212e-03	9.789648655155361e-10
3	19	4.258077196700047e+03	-8.629839649984206e-03	8.472972126693386e-10
3	20	4.455063887754256e+03	1.484804296796938e-02	7.528069568152023e-10

1.6.3 Format of a file with observational constraints:

Description:

- a collection of lines with frequency data with either (l, freq, error_bar) or (l, n, freq, error_bar) (depending on the value of `read_n` in the `AIMS_configure.py` file). For example:

```
0 1503.5 0.16
```

or the following if specifying the radial order:

```
0 15 1503.5 0.16
```

- a collection of lines with classical constraints. These start with the name of the relevant parameter (see possible options in `model.Model.string_to_param()`) followed by a description of its probability distribution function. This probability distribution function is specified in two possible ways:

- it is implicitly assumed to be Gaussian. In this situation it is only necessary to specify the mean value and the one sigma error bar. For example:

```
Teff 6100 80
```

- it is explicitly specified (different options are given in *AIMS.Distribution*):

```
Teff Uniform 6000 6200
```

- anything following a # is a comment
- the order of the lines does not matter

Examples:

- example of a file where n is *not* specified:

```
0 1582.20 0.13 # this is a (useless) comment
0 1684.02 0.16
0 1785.57 0.15
1 1526.55 0.29
1 1628.90 0.30
1 1730.45 0.17
2 1575.49 0.82
2 1676.25 0.51
2 1777.62 0.27
Teff 6060.00 84.00
Fe_H -0.20 0.09
```

- example of a file where n is specified:

```
0 15 1582.20 0.13
0 16 1684.02 0.16
Teff 6060.00 84.00 # AIMS doesn't worry about the order of the lines
0 17 1785.57 0.15
1 14 1526.55 0.29
1 15 1628.90 0.30
1 16 1730.45 0.17
2 14 1575.49 0.82
2 15 1676.25 0.51
2 16 1777.62 0.27
Fe_H -0.20 0.09
```

Differences with AMP:

- the number of frequencies does not need to be specified (if this line contains supplementary parameters, than `AIMS.py` may confuse it with frequency data)
- there are no flags (one should adjust the parameters in `AIMS_configure.py` instead)
- the order of the lines is not important (one can mix the classic and seismic observables)
- it is possible to specify radial orders (depending on the value of `read_n` in the `AIMS_configure.py` file)
- the treatment of non-seismic constraints is more flexible
 - a larger variety of non-seismic constraints can be included (see possible options in `model.Model.string_to_param()`)

- full parameter names are allowed (and preferred); for compatibility with AMP, the same one letter abbreviations are also allowed
- it is possible to specify the probability distribution function

1.7 The AIMS program

A module which contains the main program for AIMS as well as various classes which intervene when calculating the priors and likelihood function:

- *Distribution*: a class which represents a probability distribution
- *Prior_list*: a class with a list of priors
- *Mode*: a class used to represent observed modes
- *Combination*: a class used to represent frequency combinations
- *Likelihood*: a class used to represent the likelihood function
- *Probability*: a class which groups the priors and likelihood function together

This module relies on the `emcee` package to apply an MCMC algorithm which will return a representative sample of models for a given set of seismic and classic constraints.

Warning: In various places in this module, for instance in the *Prior_list* and *Likelihood* classes, various methods return what is described as a χ^2 value. Technically, these are not χ^2 values, but rather $-\chi^2/2$, i.e. the argument of the exponential function which intervenes in the Gaussian probability distribution.

`class AIMS.Combination`

A class which contains indices and coefficients which intervene in:

- linear combinations of frequencies
- frequency ratios

`add_den(j, coeff)`

Append the given index and coefficient to the list of denominator indices and coefficients.

Parameters

- `j` (*int*) – index of the mode
- `coeff` (*float*) – coefficient used in the frequency combination

`add_num(j, coeff)`

Append the given index and coefficient to the list of numerator indices and coefficients.

Parameters

- `j` (*int*) – index of the mode
- `coeff` (*float*) – coefficient used in the frequency combination

`den = None`

Value of the denominator in a frequency ratio.

`den_coeff = None`

Coefficients in the denominator of a frequency ratio, otherwise empty.

`den_index = None`

Indices in the denominator of a frequency ratio, otherwise empty.

num = None
Value of the frequency combination or numerator in a frequency ratio.

num_coeff = None
Coefficients in a linear combination or numerator of a frequency ratio.

num_index = None
Indices in a linear combination or numerator of a frequency ratio.

print_me ()
Print frequency combination.

value = None
Value of the frequency combination or ratio.

class AIMS.**Distribution** (*_type, _values*)

A class which represents a probability distribution, and can yield its value for a given input parameter, or provide a random realisation.

Parameters

- **_type** (*string*) – type of probability function (current options include “Gaussian”, “Truncated_gaussian”, “Uniform”)
- **_values** (*list of floats*) – list of parameters relevant to the probability function

print_me ()
Print type and parameters of probability distribution.

re_centre (*value*)
Re-centre the probability distribution around the input value.

Parameters value (*float*) – new value around which to centre the distribution

re_normalise (*value*)
Re-normalise the probability distribution so that its characteristic width corresponds to the input value.

Parameters value (*float*) – new value around for the characteristic width

realisation ()
Return a random value which is statistically follows the probability distribution.

Returns a random realisation

Return type float

to_string ()
Produce nice string representation of the distribution.

Returns nice string representation of the distribution

Return type string

type = None
Type of probability function (“Gaussian”, “Uniform”, or “Truncated_gaussian”)

values = None
List of parameters relevant to probability function

class AIMS.**Likelihood**

A class which described the likelihood function and allows users to evaluate it.

add_combinations (*num_list, den_list=[], target_ell=None*)

This finds the indices of modes which intervene in a frequency combination or ratio, as specified by the

mandatory and optional arguments. These indices, the relevant coefficients, the numerator, the denominator, and the resultant value of the combination are stored in the *combinations* variable.

Parameters

- **num_list** (*list of (int,int,float)*) – list of relative mode identifications and coefficients used to define a frequency combination or the numerator of a frequency ratio. This list contains tuples of the form (delta n, delta l, coeff).
- **den_list** (*list of (int,int,float)*) – list of relative mode identifications and coefficients used to define the denominator of a frequency ratio. If absent, then, it is assumed that a linear combination of frequencies is represented. The form is the same as for *num_list*.
- **target_ell** (*int*) – this is used to impose a specific l value on the first selected mode.

add_constraint (*(name, distribution)*)

Add a supplementary constraint to the list of constraints.

Parameters constraint (*(string, Distribution)*) – supplementary constraint

add_dnu_constraint (*l_targets=[0]*)

Add the large frequency separation as a constraint. The coefficients are obtained via a least-squares approach. The approach taken here has two advantages:

1. Correlations between the large frequency separation and other seismic constraints will be taken into account.
2. The same modes will be used in the same way, both for the observations and the models.

Parameters l_targets (*list of int*) – specifies for which l values the large frequency separation is to be calculated. If *None* is supplied, all modes will be used.

Note: This uses an analytical approach and is therefore the preferred method.

add_dnu_constraint_matrix (*l_targets=[0]*)

Add the large frequency separation as a constraint. The coefficients are obtained via a least-squares approach. The approach taken here has two advantages:

1. Correlations between the large frequency separation and other seismic constraints will be taken into account.
2. The same modes will be used in the same way, both for the observations and the models.

Parameters l_targets (*list of int*) – specifies for which l values the large frequency separation is to be calculated. If *None* is supplied, all modes will be used.

Note: This uses a matrix approach and is therefore *not* the preferred method.

add_nu_min_constraint (*target_ell=0, min_n=False*)

Add the minimum frequency/mode of a specific ell value as a seismic constraint. Typically, such constraints are used as an “anchor” when combined with constraints based on frequency ratios.

Parameters

- **target_ell** (*int*) – ell value of the minimum frequency/mode
- **min_n** (*boolean*) – if *False*, look for minimum observational frequency. If *True*, look for minimum radial order.

add_seismic_constraint (*string*)

Add seismic constraints based on the keyword given in *string*.

Parameters *string* (*string*) – keyword which specifies the type of constraint to be added.

Current options include:

- *nu*: individual frequencies
- *nu0*: individual frequencies (radial modes only)
- *nu_min0*: radial mode with minimum frequency
- *r02*: r_{02} frequency ratios
- *r01*: r_{01} frequency ratios
- *r10*: r_{10} frequency ratios
- *dnu*: individual large frequency separations (using all modes)
- *dnu0*: individual large frequency separations (using radial modes only)
- *avg_dnu*: average large frequency separation (using all modes)
- *avg_dnu0*: average large frequency separation (using radial modes only)

apply_constraints (*my_model*)

Calculate a χ^2 value for the set of constraints (excluding seismic constraints based on mode frequencies).

Parameters *my_model* (*model.Model*) – model for which the χ^2 value is being calculated

Returns the χ^2 value deduced from classic constraints

Return type float

assign_n (*my_model*)

Assign the radial orders based on proximity to theoretical frequencies from an input model.

Parameters *my_model* (*model.Model*) – input model

classic_weight = None

Absolute weight to be applied to classic constraints (incl. *nu_max* constraint).

clear_seismic_constraints ()

This clears the seismic constraints. Specifically, the list of seismic combinations, and associated covariance matrix and its inverse are reinitialised.

combinations = None

This contains indices and coefficients to frequency combinations and frequency ratios.

compare_frequency_combinations (*my_model*, *mode_map*, *a=[]*)

This finds a χ^2 value based on a comparison of frequencies combinations, as defined in the *combinations* variable.

Parameters

- *my_model* (*model.Model*) – model for which the χ^2 value is being calculated
- *mode_map* (*list of int*) – a mapping which relates observed modes to theoretical ones
- *a* (*array-like*) – parameters of surface correction terms

Returns the χ^2 value for the seismic constraints

Return type float

constraints = None

List of constraints which intervene in the likelihood function.

cov = None

Covariance matrix which intervenes when calculating frequency combinations.

evaluate (*my_model*)

Calculate ln of likelihood function (i.e. a χ^2 value) for a given model.

Parameters **my_model** (*model.Model*) – model for which the χ^2 value is being calculated

Returns the χ^2 value, and optionally the optimal surface amplitudes (depending on the value of `AIMS_configure.surface_option`)

Return type float, np.array (optional)

Note: This avoids model interpolation and can be used to gain time.

find_covariance ()

This prepares the covariance matrix and its inverse based on the frequency combinations in *combinations*.

Warning: This method should be called *after* all of the methods which add to the list of frequency combinations.

find_l_list (*l_targets*)

Find a list of l values with the following properties:

- each l value only occurs once
- each l value given in the parameter `l_targets` is in the result l list, except if there is 1 or less modes with this l value
- if the parameter `l_targets` is None, look for all l values with 2 or more modes associated with them

Parameters **l_targets** (*list of int*) – input list of l values

Returns new list of l values with the above properties

Return type list of int

find_map (*my_model, use_n*)

This finds a map which indicates the correspondance between observed modes and theoretical modes from *my_model*.

Parameters

- **my_model** – model for which the χ^2 value is being calculated
- **use_n** (*boolean*) – specify whether to use the radial order when finding the map from observed modes to theoretical modes. If `False`, the map is based on frequency proximity.

Returns the correspondance between observed and theoretical modes from the above model, and the number of observed modes which weren't mapped onto theoretical modes

Return type list of int, int

Note:

- a value of -1 is used to indicate that no theoretical mode corresponds to a particular observed mode.
 - only zero or one observed mode is allowed to correspond to a theoretical mode
-

find_vec (*a_combination*)

This finds a set of coefficients which intervene when constructing the covariance matrix for frequency combinations.

Parameters **a_combination** (*Combination*) – variable which specifies the frequency combination.

Returns the above set of coefficients

Return type np.array

find_weights ()

Find absolute weights for seismic and classic constraints based on options in *AIMS_configure*.

get_optimal_surface_amplitudes (*my_model*, *mode_map*)

Find optimal surface correction amplitude, for the surface correction specified by *surface_option*.

Parameters

- **my_model** (*model.Model*) – the model for which we’re finding the surface correction amplitude
- **mode_map** (*list of int*) – a mapping which relates observed modes to theoretical ones

Returns optimal surface correction amplitudes

Return type np.array

guess_dnu (*with_n=False*)

Guess the large frequency separation based on the radial modes.

Parameters **with_n** (*boolean*) – specifies whether to use the n values already stored with each mode, when calculating the large frequency separation.

Returns the large frequency separation

Return type float

guess_n ()

Guess the radial order of the observed pulsations modes.

This method uses the large frequency separation, as calculated with *guess_dnu* (), to estimate the radial orders. These orders are subsequently adjusted to avoid multiple modes with the same identification. The resultant radial orders could be off by a constant offset, but this is not too problematic when computing frequency combinations or ratios.

invcov = None

Inverse of covariance matrix, *Likelihood.cov*.

is_outside (*params*)

Test to see if the given set of parameters lies outside the grid of models. This is done by evaluate the probability and seeing if the result indicates this.

Parameters **params** (*array-like*) – input set of parameters

Returns True if the set of parameters corresponds to a point outside the grid.

Return type boolean

modes = None

List of pulsation modes (of type *Mode*).

read_constraints (*filename*, *factor=1.0*)

Read a file with pulsation data and constraints.

Parameters

- **filename** (*string*) – name of file with pulsation data.
- **factor** (*float*) – multiplicative factor for pulsation frequencies. Can be used for conversions.

seismic_weight = None

Absolute weight to be applied to seismic constraints

sort_modes ()

Sort the modes. The ordering will depend on the value of `use_n` from the `AIMS_configure.py` file.

class AIMS.**Mode** (*_n, _l, _freq, _dfreq*)

A class which describes an *observed* pulsation mode.

Parameters

- **_n** (*int*) – radial order of observed mode
- **_l** (*int*) – harmonic degree of observed mode.
- **_freq** (*float*) – pulsation frequency (in μHz).
- **_dfreq** (*float*) – error bar on pulsation frequency (in μHz).

Warning: Negative values are not accepted for `_l`, `_freq`, or `_dfreq`.

dfreq = None

Error bar on pulsation frequency (in μHz).

freq = None

Pulsation frequency (in μHz).

l = None

Harmonic degree of observed mode.

match (*a_mode*)

Check to see if input mode has the same (n,l) values as the current mode.

Parameters **a_mode** (*Mode*) – input mode which is being compared with current mode.

Returns `True` if the input mode has the same (n,l) values as the current mode.

Return type `boolean`

n = None

Radial order of observed mode.

class AIMS.**Prior_list**

A class which contains a list of priors as well as convenient methods for adding priors and for evaluating them.

add_prior (*aPrior*)

Add a prior to the list.

Parameters **aPrior** (*Distribution*) – prior which is to be added to the list.

priors = None

A list of probability distributions which correspond to priors.

realisation ()

Return a list with a realisation for each prior.

Returns a list of realisations

Return type list of floats

class AIMS.**Probability** (*_priors, _likelihood*)

A class which combines the priors and likelihood function, and allows the the user to evaluate ln of the product of these.

Parameters

- **_priors** (*Prior_list*) – input set of priors
- **_likelihood** (*Likelihood*) – input likelihood function

evaluate (*my_model*)

Evaluate the ln of the product of the priors and likelihood function, i.e. the probability, for a given model, to within an additive constant.

Parameters **my_model** (*model.Model*) – input model

Returns the ln of the probability

Return type float

Note: This avoids model interpolation and can be used to gain time.

likelihood = None

The likelihood function.

priors = None

The set of priors.

AIMS.**best_MCMC_model = None**

best model from the MCMC run

AIMS.**best_MCMC_params = None**

parameters for the model *best_MCMC_model*

AIMS.**best_MCMC_result = -1e+300**

ln(probability) result for the model *best_MCMC_model*

AIMS.**best_grid_model = None**

best model from a scan of the entire grid

AIMS.**best_grid_params = None**

parameters for the model *best_grid_model*

AIMS.**best_grid_result = -1e+300**

ln(probability) result for the model *best_grid_model*

AIMS.**check_configuration** ()

Test the values of the variables in *check_configuration* to make sure they're acceptable. If an unacceptable value is found, then this will stop AIMS and explain what variable has an erroneous value.

AIMS.**echelle_diagram** (*my_model, my_params, model_name*)

Write text file with characteristics of input model.

Parameters

- **my_model** (*model.Model*) – model for which we're writing a text file
- **my_params** (*array-like*) – parameters of the model
- **model_name** (*string*) – name used to describe this model. This is also used when naming the text file.

AIMS.**find_a_blob** (*params*)

Find a blob (i.e. supplementary output parameters) for a given set of parameters (for one model). The blob also includes the log(P) value as a first entry.

Parameters **params** (*array-like*) – input set of parameters

Returns list of supplementary output parameters

Return type list of floats

AIMS.**find_best_model** ()

Scan through grid of models to find “best” model for a given probability function (i.e. the product of priors and a likelihood function).

AIMS.**find_best_model_in_track** (*ntrack*)

Scan through an evolutionary track to find “best” model for *prob*, the probability function (i.e. the product of priors and a likelihood function).

Parameters **ntrack** (*int*) – number of the evolutionary track

Returns the ln(probability) value, and the “best” model

Return type (float, *model.Model*)

AIMS.**find_blobs** (*samples*)

Find blobs (i.e. supplementary output parameters) from a set of samples (i.e. for multiple models).

Parameters **samples** (*list/array of array-like*) – input set of samples

Returns set of supplementary output parameters

Return type np.array

AIMS.**grid = None**

grid of models

AIMS.**grid_params_MCMC = ()**

parameters used in the MCMC run (excluding surface correction parameters)

AIMS.**grid_params_MCMC_with_surf = ()**

parameters used in the MCMC run (including surface correction parameters)

AIMS.**init_walkers** ()

Initialise the walkers used in emcee.

Returns array of starting parameters

Return type np.array

AIMS.**interpolation_tests** (*filename*)

Carry out various interpolation tests and write results in binary format to file.

Parameters **filename** (*string*) – name of file in which to write test results.

Note: The contents of this file may be plotted using methods from `plot_interpolation_test.py`.

AIMS.**load_binary_data** (*filename*)

Read a binary file with a grid of models.

Parameters **filename** (*string*) – name of file with grid in binary format

Returns the grid of models

Return type *model.Model_grid*

AIMS.**log0** = **-1e+300**

a large negative value used to represent $\ln(0)$

AIMS.**my_map** = **None**

pointer to the map function (either the parallel or sequential versions)

AIMS.**ndims** = **0**

number of dimensions for MCMC parameters (includes *nsurf*)

AIMS.**nsurf** = **0**

number of surface term parameters

AIMS.**output_folder** = **None**

folder in which to write the results

AIMS.**plot_histograms** (*samples, names, fancy_names, truths=None*)

Plot a histogram based on a set of samples.

Parameters

- **samples** (*np.array*) – samples from the emcee run
- **names** (*list of strings*) – names of the quantities represented by the samples. This will be used when naming the file with the histogram
- **fancy_names** (*list of strings*) – name of the quantities represented by the samples. This will be used as the x-axis label in the histogram.
- **truths** (*list of floats*) – reference values (typically the true values or some other important values) to be added to the histograms as a vertical line

AIMS.**plot_walkers** (*samples, labels, filename, nw=3*)

Plot individual walkers.

Parameters

- **samples** (*np.array*) – samples from the emcee run
- **labels** (*list of strings*) – labels for the different dimensions in parameters space
- **filename** (*string*) – specify name of file in which to save plots of walkers.
- **nw** (*int*) – number of walkers to be plotted

Warning: This method must be applied before the samples are reshaped, and information on individual walkers lost.

AIMS.**pool** = **None**

pool from which to carry out parallel computations

AIMS.**prob** = **None**

Probability type object that represents the probability function which includes the likelihood and priors

AIMS.**run_emcee** ()

Run the emcee program.

Returns the emcee sampler for the MCMC run

AIMS.**statistical_model** = **None**

model corresponding to statistical parameters

AIMS.**statistical_params** = **None**

parameters for the model *statistical_model*

AIMS.**statistical_result** = -1e+300
ln(probability) result for the model *statistical_model*

AIMS.**string_to_title** (*string*)
Create fancy title from string.

Parameters *string* (*string*) – string from which the title is created.

Returns the fancy string title

Return type string

AIMS.**threshold** = -1e+290
threshold for “accepted” models. Needs to be greater than *log0*

AIMS.**write_LEGACY_summary** (*filename, KIC, labels, samples*)
Write a one line summary of the statistical properties based on a sequence of realisations to a file. The format matches that of the LEGACY project.

The results include:

- average values for each variable (statistical mean)
- error bars for each variable (standard mean deviation)

Parameters

- **filename** (*string*) – name of file in which to write the statistical properties
- **KIC** (*string*) – KIC number of the star
- **labels** (*list of strings*) – names of relevant variables
- **samples** (*np.array*) – samples for which statistical properties are calculated

AIMS.**write_binary_data** (*infile, outfile*)
Read an ascii file with a grid of models, and write corresponding binary file.

Parameters

- **infile** (*string*) – input ascii file name
- **outfile** (*string*) – output binary file name

AIMS.**write_combinations** (*filename, samples*)
Produce a list of linear combinations of grid models (based on interpolation) corresponding to the provided model parameters.

Parameters

- **filename** (*string*) – name of the file to which to write the model combinations
- **samples** (*np.array*) – set of model parameters for which we would like to obtain the grid models and interpolation coefficients

AIMS.**write_model** (*my_model, my_params, my_result, model_name*)
Write text file with characteristics of input model.

Parameters

- **my_model** (*model.Model*) – model for which we’re writing a text file
- **my_params** (*array-like*) – parameters of the model
- **my_result** (*float*) – ln(P) value obtained for the model

- **model_name** (*string*) – name used to describe this model. This is also used when naming the text file.

AIMS.**write_readme** (*filename*)

Write parameters relevant to this MCMC run.

Parameters **filename** (*string*) – name of file in which to write the statistical properties

AIMS.**write_samples** (*filename, labels, samples*)

Write raw samples to a file.

Parameters

- **filename** (*string*) – name of file in which to write the samples
- **labels** (*list of strings*) – names of relevant variables (used to write a header)
- **samples** (*array-like*) – samples for which statistical properties are calculated

AIMS.**write_statistics** (*filename, labels, samples*)

Write statistical properties based on a sequence of realisations to a file. The results include:

- average values for each variable (statistical mean)
- error bars for each variable (standard mean deviation)
- correlation matrix between the different variables

Parameters

- **filename** (*string*) – name of file in which to write the statistical properties
- **labels** (*list of strings*) – names of relevant variables
- **samples** (*np.array*) – samples for which statistical properties are calculated

1.8 The model module

A module which contains various classes relevant to the grid of models:

- *Model*: a model
- *Track*: an evolutionary track
- *Model_grid*: a grid of models

These different classes allow the program to store a grid of models and perform a number of operations, such as:

- retrieving model properties
- interpolate within the grid models
- sort the models within a given evolutionary track
- ...

class `model.Model` (*_glb, _name=None, _modes=None*)

A class which contains a stellar model, including classical and seismic information.

Parameters

- **_glb** (*np.array*) – 1D array of global parameters for this model. Its dimension should be greater or equal to *n_{glb}*
- **_name** (*string*) – name of the model (typically the second part of its path)

- **_modes** (*list of (int, int, float, float)*) – list of modes in the form of tuples (n,l,freq,inertia) which will be appended to the set of modes in the model.

FeH ()

Find [Fe/H] value for model.

The conversion from (x,z) to [Fe/H] is performed using the following formula:

$$[\text{Fe}/\text{H}] = \frac{[\text{M}/\text{H}]}{A_{\text{FeH}}} = \frac{1}{A_{\text{FeH}}} \log_{10} \left(\frac{z/x}{z_{\odot}/x_{\odot}} \right)$$

Returns the [Fe/H] value

Return type float

Note: The relevant values are given in *constants*

MH ()

Find [M/H] value for model.

The conversion from (x,z) to [M/H] is performed using the following formula:

$$[\text{M}/\text{H}] = \log_{10} \left(\frac{z/x}{z_{\odot}/x_{\odot}} \right)$$

Returns the [M/H] value

Return type float

Note: The relevant values are given in *constants*

append_modes (*modes*)

Append a list of modes to the model.

Parameters modes (*list of (int, int, float, float)*) – list of modes which are in the form of tuples: (n,l,freq,inertia).

cutoff ()

Find $\nu_{\text{cut-off}}$ for model.

The $\nu_{\text{cut-off}}$ value is obtained from the following scaling relation:

$$\frac{\nu_{\text{cut-off}}}{\nu_{\text{cut-off},\odot}} = \left(\frac{M}{M_{\odot}} \right) \left(\frac{R}{R_{\odot}} \right)^2 \left(\frac{T_{\text{eff}}}{T_{\text{eff},\odot}} \right)^{-1/2}$$

Returns the $\nu_{\text{cut-off}}$ value

Return type float

Note: The relevant values are given in *constants*

find_epsilon (*ltarget*)

Find epsilon, the constant offset in a simplified version of Tassoul's asymptotic formula:

$$\nu_n = \Delta\nu(n + \varepsilon)$$

Parameters ltarget (*int*) – target l value. Only modes with this l value will be used in obtaining epsilon.

Returns the constant offset

Return type float

find_large_separation ()

Find large frequency separation using only radial modes.

Returns the large frequency separation

Return type float

find_mode (*ntarget*, *ltarget*)

Find a mode with specific n and l values.

Parameters

- **ntarget** (*int*) – target n value
- **ltarget** (*int*) – target l value

Returns the frequency of the mode

Return type float

find_mode_range ()

Find n and l ranges of the modes in the model.

Returns the n and l ranges of the modes

Return type int, int, int, int

freq_sorted ()

Check to see if the frequencies are in ascending order for each l value.

Returns `True` if the frequencies are in ascending order.

Return type boolean

get_age ()

Return age of stellar model.

This is useful for sorting purposes.

Returns the age of the model

Return type float

get_freq (*surface_option=None*, *a=[]*)

Obtain model frequencies, with optional frequency corrections.

Parameters

- **surface_option** (*string*) – specifies the type of surface correction. Options include:
 - `None`: no corrections are applied
 - `"Kjeldsen2008"`: apply a correction based on Kjeldsen et al. (2008)
 - `"Ball2014"`: apply a one-term correction based on Ball and Gizon (2014)
 - `"Ball2014_2"`: apply a two-term correction based on Ball and Gizon (2014)
- **a** (*array-like*) – amplitude parameters which intervene in the surface correction

Returns models frequencies (including surface corrections)

Return type np.array

Note: If `surface_option==None` or `a==[]`, the original frequencies are returned (hence modifying them modifies the `Model` object).

get_surface_correction (*surface_option*, *a*)

Obtain corrections on model frequencies (these corrections should be *added* to the *theoretical* frequencies).

Parameters

- **surface_option** (*string*) – specifies the type of surface correction. Options include:
 - None: no corrections are applied
 - "Kjeldsen2008": apply a correction based on Kjeldsen et al. (2008)
 - "Ball2014": apply a one-term correction based on Ball and Gizon (2014)
 - "Ball2014_2": apply a two-term correction based on Ball and Gizon (2014)
- **a** (*array-like*) – parameters which intervene in the surface correction. According to the correction they take on the following meanings:
 - "Kjeldsen2008": $a[0]*freq**b_Kjeldsen2008$
 - "Ball2014": $a[0]*freq**3/I$
 - "Ball2014_2": $a[0]*freq**3/I + a[1]/(freq*I)$

Returns surface corrections on the model frequencies

Return type np.array

Note: The array operations lead to the creation of a new array with the result, which avoids modifications of the original frequencies and inertias.

glb = None

Array which will contain various global quantities

modes = None

array containing the modes (n, l, freq, inertia)

multiply_modes (*constant*)

Multiply the frequencies by constant.

Parameters constant (*float*) – constant by which the mode frequencies are multiplied

name = None

Name of the model, typically the second part of its path

numax ()

Find ν_{\max} for model.

The ν_{\max} value is obtained from the following scaling relation:

$$\frac{\nu_{\max}}{\nu_{\max,\odot}} = \left(\frac{M}{M_{\odot}}\right) \left(\frac{R}{R_{\odot}}\right)^2 \left(\frac{T_{\text{eff}}}{T_{\text{eff},\odot}}\right)^{-1/2}$$

Returns the ν_{\max} value

Return type float

Note: The relevant values are given in *constants*

print_me ()

Print classical and seismic characteristics of the model to standard output.

read_file (*filename*)

Read in a set of modes from a file.

Parameters **filename** (*string*) – name of the file with the modes. The file should contain a one-line header followed by five columns which correspond to l, n, frequency, dfreq_var, inertia.

Returns `True` if at least one frequency has been discarded (see note below).

Return type `boolean`

Note:

- The dfreq_var column is discarded.
- Frequencies above $1.1\nu_{\text{cut-off}}$ are discarded.

remove_duplicate_modes ()

Remove duplicate modes.

Modes are considered to be duplicate if they have the same l and n values (regardless of frequency).

Returns `True` if at least one mode has been removed.

Return type `boolean`

Warning: This method assumes the modes are sorted.

sort_modes ()

Sort the modes by l, then n, then freq.

string_to_param (*string*)

Return a parameter for an input string.

Parameters **string** (*string*) – string that indicates which parameter we're seeking

Returns the value of the parameter

Return type `float`

class `model.Model_grid`

A grid of models.

find_epsilons (*ltarget*)

Find epsilon values in models from the grid

Parameters **ltarget** (*int*) – target l value for which epsilons are being obtained

Returns the epsilon values

Return type list of floats

find_partition ()

Find a partition of the grid for use with `Model_grid.test_interpolation()`

Returns a random partition of [0 ... n-1] into two equal halves, where n is the number of tracks in the grid

Return type two lists of int

grid = None

Array containing the grid parameters for each evolutionary track (excluding age).

grid_params = None

Set of parameters (excluding age) used to construct the grid and do interpolations.

Note: For best interpolation results, these parameters should be comparable.

ndim = None

Number of dimensions for the grid (excluding age), as based on the `Model_grid.grid_params` variable

ndx = None

List containing track indices

plot_tessellation ()

Plot the grid tessellation.

Warning: This only works for two-dimensional tessellations.

postfix = None

Last part of the filenames which contain the model frequencies (default = ".freq").

prefix = None

Root folder with grid of models (including final slash).

read_model_list (filename)

Read list of models from a file and construct a grid.

Parameters filename (*string*) – name of the file with the list. The first line of this file should contain a prefix which is typically the root folder of the grid of models. This followed by a file with multiple columns. The first 8 contain the following information for each model:

1. the second part of the path. When concatenated with the prefix on the first line, this should give the full path to the model.
2. The stellar mass in g
3. The stellar radius in cm
4. The stellar luminosity in $\text{g.cm}^2.\text{s}^{-3}$
5. The metallicity
6. The hydrogen content
7. The stellar age in Myrs
8. The effective temperature in K

The following columns contain the parameters specified in the `AIMS_configure.user_params` variable.

tessellate ()

Apply Delaunay triangulation to obtain the grid tessellation.

tessellation = None

Object containing the tessellation of the grid used for interpolation.

test_freq ()

Test to see if frequencies in all of the models of the grid are in ascending order for each l value.

Returns

The following items are returned

- the effective temperatures of the models with frequencies out of order
- the luminosities of the models with frequencies out of order
- the effective temperatures of the models with sorted frequencies
- the luminosities of the models with sorted frequencies

Return type four lists of floats

test_interpolation ()

Test interpolation between different evolutionary tracks in a given grid.

Returns

The following four items are returned:

- the interpolation errors
- the first half of the partition (where the interpolation is tested)
- the second half of the partition (used to carry out the interpolation)
- the tessellation associated with the second half of the partition

Return type np.array, list, list, tessellation object

tracks = None

List of evolutionary tracks contained in the grid.

user_params = None

The set of user parameters involved in the grid. This is to avoid having a different set of user parameters in *AIMS_configure.py*

class model.**Track** (*aModel*, *grid_params*)

An evolutionary track.

Parameters

- **aModel** (*Model*) – first model to be added to evolutionary track (it does not need to be the youngest model in an evolutionary sequence). This Model is used to obtain the relevant parameters for the evolutionary track (as given by the *grid_params* variable).
- **grid_params** (*list of strings*) – list of strings which are the names of the parameters which describe the evolutionary track.

append (*aModel*)

Append a model to the evolutionary track.

Parameters **aModel** (*Model*) – model which is being appended to the track

find_combination (*age*, *coef*)

Return a model combination at a given age which is obtained using linear interpolation.

Parameters

- **age** (*float*) – age of desired model in Myrs
- **coef** (*float*) – coefficient which multiplies this combination

Returns pairs composed of an interpolation coefficient and a model name

Return type tuple of (float, string)

Warning: This method assumes the track is sorted, since it applies a binary search algorithm for increased efficiency.

find_mode_range ()

Find n and l ranges of modes in models

Returns the n and l ranges

Return type int, int, int, int

find_modes (*ntarget*, *ltarget*)

Return two lists, one with the ages of the models and the other with the mode frequencies corresponding to target n and l values.

This function is useful for seeing how the frequency of a particular mode changes with stellar age.

Parameters

- **ntarget** (*int*) – target n value
- **ltarget** (*int*) – target l value

Returns lists of ages and frequencies

Return type list, list

grid_params = None

Names of the parameters used to construct the grid

interpolate_model (*age*)

Return a model at a given age which is obtained using linear interpolation.

Parameters **age** (*float*) – age of desired model in Myrs

Returns the interpolated model

Return type *Model*

Warning: This method assumes the track is sorted, since it applies a binary search algorithm for increased efficiency.

is_sorted ()

Check to see if models are in ascending order according to age.

Returns `True` if the models are in order of increasing age

Return type boolean

matches (*aModel*)

Check to see if a model matches the evolutionary track and can therefore be included in the track.

Parameters **aModel** (*Model*) – input model being tested

Returns `True` only if the model given as an argument has parameters which match those of the evolutionary track.

Return type boolean

models = None

List of models in this evolutionary track

nmodes = None

Total number pulsation modes from all of the models in this evolutionary track

params = None

Parameters which characterise this evolutionary track

sort ()

Sort models within evolutionary track according to age.

test_interpolation (*nincr*)

Test accuracy of interpolation along evolutionary track.

This method removes every other model and retrieves its frequencies by interpolation from neighbouring models. The accuracy of the interpolated frequencies and global parameters are tested by carrying out comparisons with the original models.

Parameters *nincr* (*int*) – increment with which to carry out the interpolation. By comparing results for different values of *nincr*, one can evaluate how the interpolation error depends on the size of the interval over which the interpolation is carried out.

Returns the interpolation errors

Return type np.array

`model.combine_models` (*model1*, *coef1*, *model2*, *coef2*)

Do linear combination of this model with another.

This method returns a new model which is the weighted sum of two models for the purposes of model interpolation. The classical parameters are combined in a self-consistent way as are the frequencies.

Parameters

- **model1** (*Model*) – first model
- **coef1** (*float*) – weighting coefficient applied to first model
- **model2** (*Model*) – second model
- **coef2** (*float*) – weighting coefficient applied to second model

Returns the combined model

Return type *Model*

Warning: One should avoid negative or zero coefficients as these could lead to undefined results.

`model.compare_models` (*model1*, *model2*)

Compare two models and find the largest frequency different for radial and non-radial modes.

Parameters

- **model1** (*Model*) – first model
- **model2** (*Model*) – second model

Returns

a 1D array to be used in `plot_test_interpolation.py` with the following measurements of the differences between the two models:

- `result[0]` = maximum error on the radial modes
- `result[1]` = RMS error on the radial modes
- `result[2]` = RMS error on the radial modes near ν_{\max}
- `result[3]` = maximum error on the non radial modes
- `result[4]` = RMS error on the non radial modes
- `result[5]` = RMS error on the non radial modes near ν_{\max}
- `result[6+[0:n_glb]]` = errors on the global parameters

Return type np.array

`model.eps = 1e-06`

relative tolerance on parameters used for setting up evolutionary tracks

`model.find_ages (coefs, tracks, age)`

Find ages to which each track needs to be interpolated for a specified age. The global variable `scale_age` decides between the following two options:

1. `scale_age = False`: each track is simply interpolated to `age`.

2. `scale_age = True`: the age of each model along each evolutionary track, including the interpolated track, is linearly mapped onto the interval $[0,1]$. A dimensionless parameter `eta` is obtained by interpolating `age` onto the interval $[0,1]$, using the linear transformation associated with the interpolated track. Using the parameter `eta`, a corresponding age is obtained along each track.

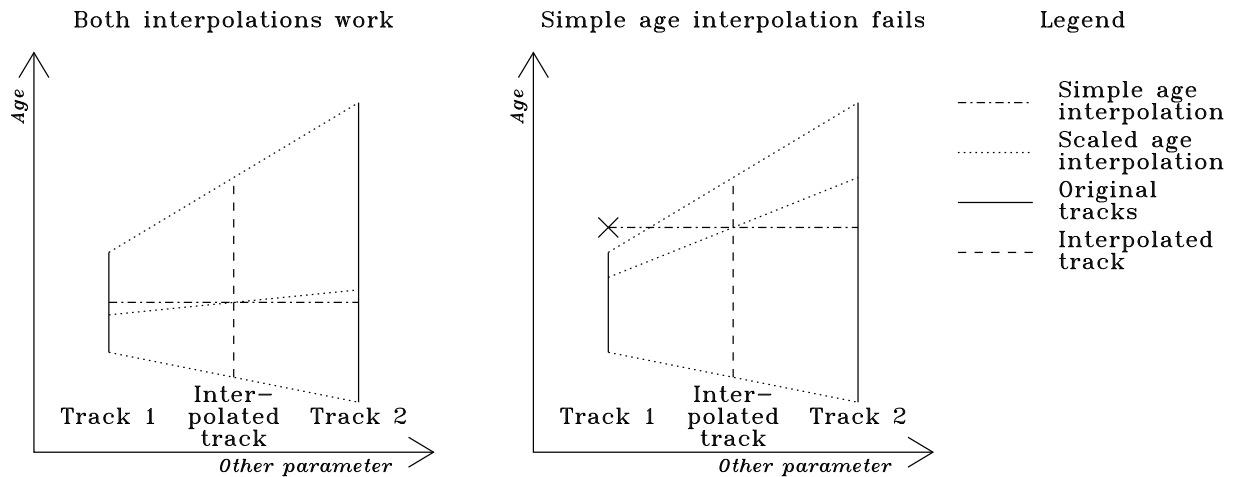


Fig. 1.1: This diagram illustrates both types of age interpolation and shows the advantages of selecting `scale_age = True`.

Parameters

- **coefs** (*list of floats*) – interpolation coefficients used to weight each track.
- **tracks** (*list of Track*) – evolutionary tracks involved in the interpolation.
- **age** (*float*) – target age for the output interpolated model.

Returns the relevant age for each track

Return type list of floats

Note:

- the interpolation coefficients should add up to 1.0
- there should be as many tracks as interpolation coefficients.

`model.find_combination (grid, pt)`

Find linear combination of models which corresponds to interpolating the model based on the provided parameters.

The interpolation is carried out using the same procedure as in `interpolate_model()`.

Parameters

- **grid** (*Model_grid*) – grid of models in which we’re carrying out the interpolation
- **pt** (*array-like*) – set of parameters used for the interpolation. The first part contains the grid parameters, whereas the last element is the age. If the provided set of parameters lies outside the grid, then `None` is returned instead of an interpolated model.

Returns pairs of coefficients and model names

Return type tuple of (float,string)

`model.find_interpolation_coefficients` (*grid, pt, tessellation, ndx*)

Find interpolation weights from the corresponding simplex.

Linear interpolation weights are obtained with the simplex by finding the barycentric coordinates of the point given by `pt`.

Parameters

- **grid** (*Model_grid*) – grid of models in which we’re carrying out the interpolation
- **pt** (*array-like*) – set of parameters used for finding the interpolation weights. The first part contains the grid parameters (relevant to this interpolation), whereas the last element is the age (not used here). If the provided set of parameters lies outside the grid, then `None` is returned instead of an interpolated model.
- **tessellation** – tessellation with which to carry out the interpolation.
- **ndx** (*list of int*) – indices of the grid points associated with the tessellation

Returns lists of interpolation coefficients and tracks

Return type list of floats, list of *Track*

`model.ftype`

type used for the frequencies

alias of `float64`

`model.get_surface_parameter_names` (*surface_option*)

Return the relevant parameter names for a given surface correction option.

Parameters **surface_option** (*string*) – specifies the type of surface correction.

Returns names for the surface parameters

Return type tuple of strings

`model.gtype`

type used for grid data

alias of `float64`

`model.iage = 0`

index of the parameter corresponding to age in the *Model.glb* array

`model.ifreq_ref = 8`

index of the parameter corresponding to the reference frequency (used to non-dimensionalise the pulsation frequencies of the model) in the *Model.glb* array

`model.iluminosity = 10`

index of the parameter corresponding to luminosity in the *Model.glb* array

`model.imass = 1`

index of the parameter corresponding to mass in the *Model.glb* array

`model.init_user_param_dict()`

Initialise the dictionaries which are related to user-defined parameters. For a given parameter, these dictionaries provide the appropriate index for for the `Model.glb` array as well as the appropriate latex name.

`model.interpolate_model(grid, pt, tessellation, ndx)`

Interpolate model in grid using provided parameters.

The interpolation is carried out in two steps. First, linear interpolation according to age is carried out on each node of the simplex containing the set of parameters. This interpolation is done using the `Track.interpolate_model` method. Then, linear interpolation is carried out within the simplex. This achieved by finding the barycentric coordinates of the model (i.e. the weights), before combining the age-interpolated models from the nodes using the `combine_models` method. In this manner, the weights are only calculated once, thereby increasing computational efficiency.

Parameters

- **grid** (`Model_grid`) – grid of models in which we’re carrying out the interpolation
- **pt** (*array-like*) – set of parameters used for the interpolation. The first part contains the grid parameters, whereas the last element is the age. If the provided set of parameters lies outside the grid, then `None` is returned instead of an interpolated model.
- **tessellation** – tessellation with which to carry out the interpolation.
- **ndx** (*list of int*) – indices of the grid points associated with the tessellation

Returns the interpolated model

Return type `Model`

`model.iradius = 9`

index of the parameter corresponding to radius in the `Model.glb` array

`model.itemperature = 2`

index of the parameter corresponding to temperature in the `Model.glb` array

`model.ix0 = 4`

index of the parameter corresponding to the initial hydrogen content in the `Model.glb` array

`model.iz0 = 3`

index of the parameter corresponding to the initial metallicity the `Model.glb` array

`model.ltype`

type used for the l values

alias of `int8`

`model.modetype = [('n', <type 'numpy.int16'>), ('l', <type 'numpy.int8'>), ('freq', <type 'numpy.float64'>), ('inertia', <type`

structure for modes

`model.nglb = 11`

total number of global quantities in a model (see `Model.glb`).

`model.nlin = 8`

total number of global quantities which are interpolated in a linear way (see `combine_models()`). These quantities are numbered 0:nlin-1

`model.ntype`

type used for the n values

alias of `int16`

`model.string_to_latex(string, prefix='', postfix='')`

Return a fancy latex name for an input string.

Parameters

- **string** (*string*) – string that indicates for which parameter we’re seeking a latex name
- **prefix** (*string*) – optional prefix to add to the string
- **postfix** (*string*) – optional postfix to add to the string

Returns a fancy latex name

Return type string

Note: This also works for the names of the amplitude parameters for surface corrections.

`model.tol = 1e-06`

tolerance level for points slightly outside the grid

`model.user_params_index = {}`

dictionary which will supply the appropriate index for the user-defined parameters

`model.user_params_latex = {}`

dictionary which will supply the appropriate latex name for the user-defined parameters

1.9 The constants module

A module which contains the following physical constants:

Name of variable	Quantity it describes	Units
<i>solar_radius</i>	the solar radius	cm
<i>solar_mass</i>	the solar mass	g
<i>solar_luminosity</i>	the solar luminosity	$\text{g}\cdot\text{cm}^2\cdot\text{s}^{-3}$
<i>solar_temperature</i>	the solar effective temperature	K
<i>solar_dnu</i>	the solar large frequency separation	μHz
<i>solar_numax</i>	the solar frequency at maximum power	μHz
<i>solar_cutoff</i>	the solar cutoff frequency	μHz
<i>G</i>	the gravitational constant	$\text{cm}^3\cdot\text{g}^{-1}\cdot\text{s}^{-2}$
<i>solar_x</i>	the solar hydrogen content	dimensionless
<i>solar_z</i>	the solar metallicity content	dimensionless
<i>A_FeH</i>	multiplicative constant in $[\text{M}/\text{H}] = A_{\text{FeH}}[\text{Fe}/\text{H}]$	dimensionless

Note: These values can be edited according to the latest discoveries. As good practise, it is helpful to include the relevant reference.

`constants.A_FeH = 1.0`

multiplicative constant which intervenes in the conversion from metal content to iron content

`constants.G = 6.67428e-08`

the gravitational constant in $\text{cm}^3\cdot\text{g}^{-1}\cdot\text{s}^{-2}$

`constants.solar_cutoff = 5300.0`

the solar cut-off frequency separation in μHz

`constants.solar_dnu = 135.1`

the solar large frequency separation in μHz

`constants.solar_luminosity = 3.844e+33`

the solar luminosity in $\text{g}\cdot\text{cm}^2\cdot\text{s}^{-3}$

`constants.solar_mass = 1.98919e+33`
the solar mass in g

`constants.solar_numax = 3090.0`
the solar frequency at maximum power in μHz

`constants.solar_radius = 6959900000.0`
the solar radius in cm

`constants.solar_temperature = 5777.0`
the solar temperature in K

`constants.solar_x = 0.7336`
the solar hydrogen content

`constants.solar_z = 0.0179`
the solar metallicity content

1.10 The utilities module

A module which contains various utility methods for handling strings and floats.

`utilities.is_number(s)`

Test a string to see if it is a number.

Parameters *s* (*string*) – string which is being tested

Returns `True` if *s* is a number, and `False` otherwise

Return type `boolean`

Note: This method allows “d” and “D” as an exponent (i.e. for Fortran style numbers).

`utilities.to_float(s)`

Convert a string to a float.

Parameters *s* (*string*) – string which will be converted to a float

Returns the corresponding float

Return type `float`

Note: This method allows “d” and “D” as an exponent (i.e. for Fortran style numbers).

`utilities.trim(s)`

Return a string with comments (starting with “#”) removed.

Parameters *s* (*string*) – the string for which we would like to remove comments.

Returns the string without comments

Return type `string`

1.11 The ImprovedTessellation module

This implements a *ImprovedTessellation*, a subclass of `scipy.spatial.Delaunay` which provides support for extrapolation outside the grid of points, as well as various utility methods.

class ImprovedTessellation.**ImprovedTessellation** (*points*)

A class which extends `scipy.spatial.Delaunay` thereby providing an improved tessellation which allows the user to find the nearest simplex outside the convex hull of the grid of points upon which the tessellation is built. This facilitates extrapolation outside the grid of points.

Parameters *points* (*(npoints, ndim) numpy float array*) – coordinates of the points to triangulate

convex_hull_array = None

Array representation of the facets which make up the convex hull

facet_to_simplex = None

Lookup array which gives the index of the simplex corresponding to each facet from the convex hull

find_barycentre_distances ()

Find minimum and maximum distances between the barycentres of the simplices and points which make up these simplices.

Returns the set of distances

Return type 2D numpy float array

find_barycentres ()

Find the barycentres of the simplices.

Returns the set of barycentres

Return type 2D numpy float array

find_distances (*points*)

Find minimum and maximum distances between a given set of points, and the points of the closest simplices.

Parameters *points* (*(npoints, ndim) numpy float array*) – coordinates of the points for which we would like to find the maximum and minimum distances

Returns the set of distances

Return type 2D numpy float array

find_simplex (*pt*)

Find the simplices containing the given points. If the given point is outside the convex hull of the grid of points, find nearest simplex with the most favourable height.

Parameters *pt* (*array-like of floats*) – point for which we're searching for the containing or nearest simplex

Returns the index of the containing or nearest simplex

Return type int

find_simplex_volumes ()

Find volumes of simplices.

Returns the set of volumes

Return type 1D numpy float array

static point_simplex_distance (*pt, simplex*)

Recursive function which finds the distance between a simplex and a point. Reference: [Golubitsky, O., Mazalov, V., Watt, S. M. \(2015\)](#)

Parameters

- **pt** (*numpy 1D float array*) – n-dimensional point
- **simplex** (*(d,n) numpy array*) – a (d-1)-dimensional simplex (d-1 <= n)

Returns the distance between `pt` and the closest point in `simplex`

Return type float

`simplex_to_simplex_array` (*val*)

Convert an index representing a simplex to an array representation.

Parameters `val` (*integer*) – index corresponding to a simplex

Returns an array representation of the simplex

Return type (ndim+1,ndim) numpy float array

static `simplex_volume` (*simplex*)

Calculate the volume of a simplex. The approach used here also works for simplices embedded in higher dimensional spaces (e.g. a triangle in 3D space).

Parameters `simplex` (*(d,n) numpy array*) – a (d-1)-dimensional simplex (d-1 <= n)

Returns the volume of the simplex

Return type float

1.12 The `plot_interpolation_test` tool

An interactive utility which plots various forms of interpolation error, stored in a binary file produced by `AIMS.test_interpolation()`. It specifically tests the errors from two types of interpolation:

- age interpolation: this is interpolation along a given evolutionary track
- track interpolation: this is interpolation between different evolutionary tracks

This utility allows various types of plots:

- 3D plots of interpolation errors as a function of grid structural parameters
- 2D slices which show interpolation errors as a function of age for a given evolutionary track
- interactive plots which allow you to select 2D slices

Note: Interpolation errors for models in a given evolutionary track are typically stored in arrays as follows:

- `result[model_number, ndim+0]` = maximum error on the radial modes
- `result[model_number, ndim+1]` = RMS error on the radial modes
- `result[model_number, ndim+2]` = RMS error on the radial modes near ν_{\max}
- `result[model_number, ndim+3]` = maximum error on the non radial modes
- `result[model_number, ndim+4]` = RMS error on the non radial modes
- `result[model_number, ndim+5]` = RMS error on the non radial modes near ν_{\max}
- `result[model_number, ndim+6+[0:n_glb]]` = errors on the global parameters

where:

- `result` = the array which contains the interpolation errors
- `model_number` = an index which represents the model (not necessarily the number of the model along the evolutionary track)
- `ndim` = the number of dimensions in the grid (including age)
- `n_glb` = the number of global parameters for stellar models in the grid

`plot_interpolation_test.all_nan` (*array*)

Test to see if all of the elements of an array are nan's.

Parameters `array` (*np.array*) – array in which we're checking to see if all elements are nan's.

Returns True if all the elements of `array` are nan's, and False otherwise.

Return type boolean

`plot_interpolation_test.ndim = 0`

number of dimension in grid (including age)

`plot_interpolation_test.nglb = 0`

number of global parameters

`plot_interpolation_test.onpick_age` (*event*)

Event catcher for the grid plot (which shows the positions of the evolutionary tracks as a function of the grid parameters, excluding age).

Parameters:

Parameters `event` – event caught by the grid plot.

`plot_interpolation_test.onpick_track` (*event*)

Event catcher for the partition tessellation plot (associated with tests of track interpolation).

Parameters `event` – event caught by the partition tessellation plot.

`plot_interpolation_test.plot3D` (*results, error_ndx, tpe='max', title=None, truncate=0*)

Create 3D plot showing the error as a function of the two first grid parameters.

Parameters

- **results** (*list of np.arrays*) – list of 2D arrays which contain various types of errors as a function of the model number along a given evolutionary track.
- **error_ndx** (*int*) – value which specifies the type of error to be plotted.
- **tpe** (*string*) – specifies how to combine errors along the evolutionary track. Options include:
 - “max”: corresponds to taking the maximum value.
 - “avg”: takes the mean-square value.
- **title** (*string*) – the title of the plot
- **truncate** (*int*) – (default = 0): specifies how many models should be omitted on both ends of the track. This is useful for comparing results from tests involving different sizes of increments.

Note: See above introductory description for a more detailed description of the indices which intervene in the 2D arrays contained in `results` and of the relevant values for `error_ndx`.

`plot_interpolation_test.plot_grid` (*grid*)

Make an interactive plot of the grid. Clicking on the blue dots will produce a 2D slice showing age interpolation errors for the associated evolutionary track.

Parameters `grid` (*np.array*) – array containing basic grid parameters (excluding age)

Warning: This only works for two-dimensional grids (excluding the age dimension).

`plot_interpolation_test.plot_partition_tessellation` (*grid*, *ndx1*, *ndx2*, *tessellation*)

Make an interactive tessellation plot based on the supplied partition on the grid. Clicking on the blue dots will produce a 2D slice showing track interpolation errors for the associated evolutionary track.

Parameters

- **grid** (*np.array*) – array containing basic grid parameters (excluding age)
- **ndx1** (*list of int*) – list with the indices of the first part of the partition.
- **ndx2** (*list of int*) – list with the indices of the second part of the partition.
- **tessellation** – grid tessellation associated with `ndx2`

Warning: This only works for two-dimensional tessellations.

`plot_interpolation_test.plot_slice_age` (*pos*)

Plot age interpolation error as a function of age for a given track.

Parameters **pos** (*int*) – index of the relevant track.

Note: This *pos* index applies to `results_age`, i.e., it is based on the original track indices.

`plot_interpolation_test.plot_slice_track` (*pos*)

Plot track interpolation error as a function of age for a given track.

Parameters **pos** (*int*) – index of the relevant track.

Note: This *pos* index applies to `results_track`, i.e., it is based on the indices deduced from the grid partition.

`plot_interpolation_test.results_age = None`

list which contains the arrays with the errors from age interpolation

`plot_interpolation_test.results_track = None`

list which contains the arrays with the errors from track interpolation

`plot_interpolation_test.titles = None`

the grid quantities, which will serve as axis labels

INDICES AND TABLES

- genindex
- modindex
- search

a

AIMS, 8

c

constants, 31

i

ImprovedTessellation, 32

m

model, 19

p

plot_interpolation_test, 34

u

utilities, 32

A

A_FeH (in module constants), 31
 add_combinations() (AIMS.Likelihood method), 9
 add_constraint() (AIMS.Likelihood method), 10
 add_den() (AIMS.Combination method), 8
 add_dnu_constraint() (AIMS.Likelihood method), 10
 add_dnu_constraint_matrix() (AIMS.Likelihood method), 10
 add_nu_min_constraint() (AIMS.Likelihood method), 10
 add_num() (AIMS.Combination method), 8
 add_prior() (AIMS.Prior_list method), 14
 add_seismic_constraint() (AIMS.Likelihood method), 10
 AIMS (module), 8
 all_nan() (in module plot_interpolation_test), 35
 append() (model.Track method), 25
 append_modes() (model.Model method), 20
 apply_constraints() (AIMS.Likelihood method), 11
 assign_n() (AIMS.Likelihood method), 11

B

best_grid_model (in module AIMS), 15
 best_grid_params (in module AIMS), 15
 best_grid_result (in module AIMS), 15
 best_MCMC_model (in module AIMS), 15
 best_MCMC_params (in module AIMS), 15
 best_MCMC_result (in module AIMS), 15

C

check_configuration() (in module AIMS), 15
 classic_weight (AIMS.Likelihood attribute), 11
 clear_seismic_constraints() (AIMS.Likelihood method), 11
 Combination (class in AIMS), 8
 combinations (AIMS.Likelihood attribute), 11
 combine_models() (in module model), 27
 compare_frequency_combinations() (AIMS.Likelihood method), 11
 compare_models() (in module model), 27
 constants (module), 31
 constraints (AIMS.Likelihood attribute), 11
 convex_hull_array (ImprovedTessellation.ImprovedTessellation attribute), 33

cov (AIMS.Likelihood attribute), 11
 cutoff() (model.Model method), 20

D

den (AIMS.Combination attribute), 8
 den_coeff (AIMS.Combination attribute), 8
 den_index (AIMS.Combination attribute), 8
 dfreq (AIMS.Mode attribute), 14
 Distribution (class in AIMS), 9

E

echelle_diagram() (in module AIMS), 15
 eps (in module model), 27
 evaluate() (AIMS.Likelihood method), 12
 evaluate() (AIMS.Probability method), 15

F

facet_to_simplex (ImprovedTessellation.ImprovedTessellation attribute), 33
 FeH() (model.Model method), 20
 find_a_blob() (in module AIMS), 15
 find_ages() (in module model), 28
 find_barycentre_distances() (ImprovedTessellation.ImprovedTessellation method), 33
 find_barycentres() (ImprovedTessellation.ImprovedTessellation method), 33
 find_best_model() (in module AIMS), 16
 find_best_model_in_track() (in module AIMS), 16
 find_blobs() (in module AIMS), 16
 find_combination() (in module model), 28
 find_combination() (model.Track method), 25
 find_covariance() (AIMS.Likelihood method), 12
 find_distances() (ImprovedTessellation.ImprovedTessellation method), 33
 find_epsilon() (model.Model method), 20
 find_epsilons() (model.Model_grid method), 23
 find_interpolation_coefficients() (in module model), 29
 find_l_list() (AIMS.Likelihood method), 12
 find_large_separation() (model.Model method), 21
 find_map() (AIMS.Likelihood method), 12
 find_mode() (model.Model method), 21
 find_mode_range() (model.Model method), 21

find_mode_range() (model.Track method), 25
 find_modes() (model.Track method), 26
 find_partition() (model.Model_grid method), 23
 find_simplex() (ImprovedTessellation.ImprovedTessellation method), 33
 find_simplex_volumes() (ImprovedTessellation.ImprovedTessellation method), 33
 find_vec() (AIMS.Likelihood method), 12
 find_weights() (AIMS.Likelihood method), 13
 freq (AIMS.Mode attribute), 14
 freq_sorted() (model.Model method), 21
 ftype (in module model), 29

G

G (in module constants), 31
 get_age() (model.Model method), 21
 get_freq() (model.Model method), 21
 get_optimal_surface_amplitudes() (AIMS.Likelihood method), 13
 get_surface_correction() (model.Model method), 21
 get_surface_parameter_names() (in module model), 29
 glb (model.Model attribute), 22
 grid (in module AIMS), 16
 grid (model.Model_grid attribute), 23
 grid_params (model.Model_grid attribute), 23
 grid_params (model.Track attribute), 26
 grid_params_MCMC (in module AIMS), 16
 grid_params_MCMC_with_surf (in module AIMS), 16
 gtype (in module model), 29
 guess_dnu() (AIMS.Likelihood method), 13
 guess_n() (AIMS.Likelihood method), 13

I

iage (in module model), 29
 ifreq_ref (in module model), 29
 iluminosity (in module model), 29
 imass (in module model), 29
 ImprovedTessellation (class in ImprovedTessellation), 32
 ImprovedTessellation (module), 32
 init_user_param_dict() (in module model), 29
 init_walkers() (in module AIMS), 16
 interpolate_model() (in module model), 30
 interpolate_model() (model.Track method), 26
 interpolation_tests() (in module AIMS), 16
 invcov (AIMS.Likelihood attribute), 13
 iradius (in module model), 30
 is_number() (in module utilities), 32
 is_outside() (AIMS.Likelihood method), 13
 is_sorted() (model.Track method), 26
 itemperature (in module model), 30
 ix0 (in module model), 30
 iz0 (in module model), 30

L

l (AIMS.Mode attribute), 14
 likelihood (AIMS.Probability attribute), 15
 Likelihood (class in AIMS), 9
 load_binary_data() (in module AIMS), 16
 log0 (in module AIMS), 16
 ltype (in module model), 30

M

match() (AIMS.Mode method), 14
 matches() (model.Track method), 26
 MH() (model.Model method), 20
 Mode (class in AIMS), 14
 Model (class in model), 19
 model (module), 19
 Model_grid (class in model), 23
 models (model.Track attribute), 26
 modes (AIMS.Likelihood attribute), 13
 modes (model.Model attribute), 22
 modetype (in module model), 30
 multiply_modes() (model.Model method), 22
 my_map (in module AIMS), 17

N

n (AIMS.Mode attribute), 14
 name (model.Model attribute), 22
 ndim (in module plot_interpolation_test), 35
 ndim (model.Model_grid attribute), 24
 ndims (in module AIMS), 17
 ndx (model.Model_grid attribute), 24
 nglb (in module model), 30
 nglb (in module plot_interpolation_test), 35
 nlin (in module model), 30
 nmodes (model.Track attribute), 26
 nsurf (in module AIMS), 17
 ntype (in module model), 30
 num (AIMS.Combination attribute), 8
 num_coeff (AIMS.Combination attribute), 9
 num_index (AIMS.Combination attribute), 9
 numax() (model.Model method), 22

O

onpick_age() (in module plot_interpolation_test), 35
 onpick_track() (in module plot_interpolation_test), 35
 output_folder (in module AIMS), 17

P

params (model.Track attribute), 26
 plot3D() (in module plot_interpolation_test), 35
 plot_grid() (in module plot_interpolation_test), 35
 plot_histograms() (in module AIMS), 17
 plot_interpolation_test (module), 34

plot_partition_tessellation() (in module plot_interpolation_test), 35
 plot_slice_age() (in module plot_interpolation_test), 36
 plot_slice_track() (in module plot_interpolation_test), 36
 plot_tessellation() (model.Model_grid method), 24
 plot_walkers() (in module AIMS), 17
 point_simplex_distance() (ImprovedTessellation.ImprovedTessellation static method), 33
 pool (in module AIMS), 17
 postfix (model.Model_grid attribute), 24
 prefix (model.Model_grid attribute), 24
 print_me() (AIMS.Combination method), 9
 print_me() (AIMS.Distribution method), 9
 print_me() (model.Model method), 22
 Prior_list (class in AIMS), 14
 priors (AIMS.Prior_list attribute), 14
 priors (AIMS.Probability attribute), 15
 prob (in module AIMS), 17
 Probability (class in AIMS), 14

R

re_centre() (AIMS.Distribution method), 9
 re_normalise() (AIMS.Distribution method), 9
 read_constraints() (AIMS.Likelihood method), 13
 read_file() (model.Model method), 22
 read_model_list() (model.Model_grid method), 24
 realisation() (AIMS.Distribution method), 9
 realisation() (AIMS.Prior_list method), 14
 remove_duplicate_modes() (model.Model method), 23
 results_age (in module plot_interpolation_test), 36
 results_track (in module plot_interpolation_test), 36
 run_emcee() (in module AIMS), 17

S

seismic_weight (AIMS.Likelihood attribute), 14
 simplex_to_simplex_array() (ImprovedTessellation.ImprovedTessellation method), 34
 simplex_volume() (ImprovedTessellation.ImprovedTessellation static method), 34
 solar_cutoff (in module constants), 31
 solar_dnu (in module constants), 31
 solar_luminosity (in module constants), 31
 solar_mass (in module constants), 31
 solar_numax (in module constants), 32
 solar_radius (in module constants), 32
 solar_temperature (in module constants), 32
 solar_x (in module constants), 32
 solar_z (in module constants), 32
 sort() (model.Track method), 26
 sort_modes() (AIMS.Likelihood method), 14
 sort_modes() (model.Model method), 23
 statistical_model (in module AIMS), 17

statistical_params (in module AIMS), 17
 statistical_result (in module AIMS), 17
 string_to_latex() (in module model), 30
 string_to_param() (model.Model method), 23
 string_to_title() (in module AIMS), 18

T

tessellate() (model.Model_grid method), 24
 tessellation (model.Model_grid attribute), 24
 test_freq() (model.Model_grid method), 24
 test_interpolation() (model.Model_grid method), 25
 test_interpolation() (model.Track method), 27
 threshold (in module AIMS), 18
 titles (in module plot_interpolation_test), 36
 to_float() (in module utilities), 32
 to_string() (AIMS.Distribution method), 9
 tol (in module model), 31
 Track (class in model), 25
 tracks (model.Model_grid attribute), 25
 trim() (in module utilities), 32
 type (AIMS.Distribution attribute), 9

U

user_params (model.Model_grid attribute), 25
 user_params_index (in module model), 31
 user_params_latex (in module model), 31
 utilities (module), 32

V

value (AIMS.Combination attribute), 9
 values (AIMS.Distribution attribute), 9

W

write_binary_data() (in module AIMS), 18
 write_combinations() (in module AIMS), 18
 write_LEGACY_summary() (in module AIMS), 18
 write_model() (in module AIMS), 18
 write_readme() (in module AIMS), 19
 write_samples() (in module AIMS), 19
 write_statistics() (in module AIMS), 19